Animation through Transformations

Adhel-Esteban Rivera
adhel.rivera2@upr.edu
Universidad de Puerto Rico | Recinto Universitario de Mayagüez
Mayaguez, Puerto Rico, USA

Abstract

In *Computer Graphics from Scratch* a Ray Tracer is built using linear algebra concepts to produce static images of 3 spheres, however, this Ray tracer is not capable of motion. This paper will explain how motion and new primitives were implemented into this Ray Tracer, and the challenges and optimizations using Linear Algebra, OpenMP and C++ principles to result in fast and accurate sequential image generation to create animation.

CCS Concepts

• **Computing methodologies** → **Ray tracing**.

Keywords

Ray Tracing, 3D Rendering, Computer Graphics, OpenMP, Parallel Processing, Matrix Transformations, Object Rotation, Animation, Scene Generation, Primitives, Camera Motion, 3D Scene Manipulation, Real-Time Rendering, Render Optimization, Vector Mathematics, Transformation Matrices, Object Positioning, GPU Rendering, Performance Benchmarking, Tetrahedron, Cube Rendering, Rotation Matrices, Frame Rendering, Motion Animation, Dolly Shot, Render Time Comparison, Image Sequencing

ACM Reference Format:
Adhel-Esteban Rivera. 2024. Animation through Transformations. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'COMP)*. COMP, Mayaguez, PR, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Historically, animation has always been done by running a sequence of slightly different photos one after the other [**3**]. Since the first animations in a zoetrope, to Walt Disney's caricatures, and Pixar's CG films, the concept is the same: slightly different photos changed in sequence one after the other. All of these running in short sequence once after the other essentially "trick" the brain into seeing it as motion. The sweet spot in Hollywood has always been 24 photos per second, and for smoother animations like in video games, 60 photos per second has been the norm. Each of these "photos" in animations is referred to as a "frame", hence the rate of photos run per second is "fps", which is frames per second.

Conference acronym 'COMP, December 12-13, 2024, Mayaguez, PR

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

The Ray Tracer done in the book *Computer Graphics From Scratch* highlights the key aspects of Ray Tracing which include intersections and light calculations using linear algebra theory resulting in clear and realistic spheres in a scene with different colors and different light interactions[**1**]. However, when it came to having more primitive shapes like cubes and tetrahedrons, or having motion, the system was quite limited, and required some heavy refactoring and algorithm optimizations to add new primitives and bring these primitives to life in a fast way.

2 Adding Primitives

The Ray Tracer from *Computer Graphics From Scratch* only covers the process of making spheres using sphere formulas. When it comes down to adding primitive shapes such as a Cube or Tetrahedron, the current tools in the Ray Tracing arsenal of making spheres aren't enough. In Computer Graphics, everything can be made out of triangles because of the specific properties the triangles have.

1. **Planarity**: Triangles are the simplest way to define a plane and are always planar because of the three vertices used. Any three points in a space define a plane.
2. **Simplicity**: Triangles have the fewest necessary vertices and edges to define a polygon.
3. **Versatility**: Cubes can be made up of triangles as each face requires two triangles, and tetrahedrons can also be defined with triangles. Ultimately, any shape can be defined using only triangles.

This is why triangles are used to define any shape in Computer Graphics. For this one, we'll focus on two primitives: a cube, and a tetrahedron.

2.1 Defining Triangles in Ray Tracer

Any 3 points in a 3-D space define a plane. But when it comes to rendering only what is within those three points, a check has to be run for when a ray is "shot" from the camera to check if that ray intersects with the triangle within the 3-D plane. If the ray is within that triangle space, it returns the triangle color. This is done using Barycentric Coordinates which calculate the relative weights of a point with respect to the edges of a triangle.

Vectors:

$$\mathbf{v_0} = \mathbf{C} - \mathbf{A} \quad \text{(Edge from } A \text{ to } C\text{)},$$
$$\mathbf{v_1} = \mathbf{B} - \mathbf{A} \quad \text{(Edge from } A \text{ to } B\text{)},$$
$$\mathbf{v_2} = \mathbf{P} - \mathbf{A} \quad \text{(Vector from } A \text{ to } P\text{)}.$$

Dot Products:

$$\begin{aligned}
\text{dot00} &= \mathbf{v_0} \cdot \mathbf{v_0} \quad \text{(Squared length of } \mathbf{v_0}\text{)},\\
\text{dot01} &= \mathbf{v_0} \cdot \mathbf{v_1} \quad \text{(Relationship between } \mathbf{v_0} \text{ and } \mathbf{v_1}\text{)},\\
\text{dot02} &= \mathbf{v_0} \cdot \mathbf{v_2} \quad \text{(Relationship between } \mathbf{v_0} \text{ and } \mathbf{v_2}\text{)},\\
\text{dot11} &= \mathbf{v_1} \cdot \mathbf{v_1} \quad \text{(Squared length of } \mathbf{v_1}\text{)},\\
\text{dot12} &= \mathbf{v_1} \cdot \mathbf{v_2} \quad \text{(Relationship between } \mathbf{v_1} \text{ and } \mathbf{v_2}\text{)}.
\end{aligned}$$

Using the dot products, we compute barycentric coordinates u and v, which represent the relative weights of P with respect to the edges $\mathbf{v_0}$ and $\mathbf{v_1}$:

$$u = \frac{\text{dot11} \cdot \text{dot02} - \text{dot01} \cdot \text{dot12}}{\text{dot00} \cdot \text{dot11} - \text{dot01}^2},$$
$$v = \frac{\text{dot00} \cdot \text{dot12} - \text{dot01} \cdot \text{dot02}}{\text{dot00} \cdot \text{dot11} - \text{dot01}^2}.$$

Here: u represents how far P is along $\mathbf{v_1}$, and v represents how far P is along $\mathbf{v_0}$.

The point P is inside the triangle if:

$$\begin{aligned}
u &\geq 0 \quad (P \text{ is not outside edge } AB),\\
v &\geq 0 \quad (P \text{ is not outside edge } AC),\\
u + v &\leq 1 \quad (P \text{ is not outside edge } BC).
\end{aligned}$$

Implemented in code:

Listing 1: Point in Triangle Code

```
    Vector3D A = triangle.getPointA();
    Vector3D B = triangle.getPointB();
    Vector3D C = triangle.getPointC();

    Vector3D v0 = C - A;
    Vector3D v1 = B - A;
    Vector3D v2 = P - A;

    double dot00 = v0 * v0;
    double dot01 = v0 * v1;
    double dot02 = v0 * v2;
    double dot11 = v1 * v1;
    double dot12 = v1 * v2;

    double invDenom = 1.0/(dot00*dot11-dot01*dot01);
    double u = (dot11 * dot02 - dot01 * dot12) * invDenom
        ;
    double v = (dot00 * dot12 - dot01 * dot02) * invDenom
        ;

    return (u >= 0) && (v >= 0) && (u + v <= 1);
}
```

2.2 Defining Primitives in Using Triangles

Once a triangle is defined, it is simple to create primitive objects with triangles. A cube consists of 6 faces, each with 4 shared vertices, which results in 8 vertices total. Figure 1 below serves as an example of how triangles can be distributed on the faces of a cube to form it:

Imgs/Cube Design.png

Figure 1: How randomly colored triangles can be distributed on the surfaces of a cube.

The cube is defined with only triangles as seen in Figure 1. Each vertex point is the vertex point of another triangle, so order is important. Two triangles define a face, and the are sharing vertices.

To define the location of the cube, a position vector $\vec{P}$ was used which points to the center of the cube. The vertices $\vec{V}$ of the cube were defined by taking a parameter *width* and adding it to the position vectors.

In mathematical form, the vertex vector $\vec{V}$ is defined by taking the sum of the position vector $\vec{P}$ and the selectedVertex additions depending which side of the cube is defined.

$$\vec{V} = \vec{P} + \vec{W_{SelectedVertex}}$$

In the code, each vertex is defined as *topCorner* or *bottomCorner*, and the first letter stands for *left* or *right*, and the second for *front* or *rear*.

Listing 2: Cube Generation C++ Code

```
    // Top Plane Coordinates
    // Top Left Rear Corner
    Vector3D topCornerLR = (Vector3D(center.getX()-width,
          center.getY()+width, center.getZ()-width));

    // Top Right Rear Corner
    Vector3D topCornerRR = (Vector3D(center.getX()+width,
          center.getY()+width, center.getZ()-width));

    repeat for all corners.


    // Top Triangles
    Primitive t1 = Primitive(color, topCornerLR,
         topCornerRR, topCornerLF, specular, reflective);
    Primitive t2 = Primitive(color, topCornerRR,
         topCornerLF, topCornerRF, specular, reflective);
```

```
    // Bottom Triangles
    Primitive t3 = Primitive(color, bottomCornerLR,
        bottomCornerRR, bottomCornerLF, specular,
        reflective);
    Primitive t4 = Primitive(color, bottomCornerRR,
        bottomCornerLF, bottomCornerRF, specular,
        reflective);

    Repeat for each triangle.

    array<Primitive, 12> triangles = {t1, t2, t3, t4, t5,
        t6, t7, t8, t9, t10, t11, t12};
```

The same theory applies for tetrahedrons, although it is simpler because there are less vertices to calculate.

Listing 3: Tetrahedron Generation C++ Code

```
    Matrix3D rotationMatrix = getRotationMatrix(
        angleInput*1, angleInput*2, angleInput*3);

    Vector3D pointA = (Vector3D(center.getX(), center.
        getY(), center.getZ()-width)); // Tip

    Repeat for each vertex

    Primitive t1 = Primitive(Vector3D(255, 0, 0), pointA,
        pointB, pointC, specular, reflective);

    Repeat for each face

    array<Primitive, 4> triangles = {t1, t2, t3, t4};
    return triangles;
}
```

When it comes to defining rotation, this was more complicated. The sphere did not need rotation because it is uniform throughout and a rotation wouldn't be seen even. The other primitives do need rotation because it is very noticeable. This provided a bit of technical cleverness because each vertex needed to spin around the primitive's own axis. A transformation matrix was used to accomplish this task. The following represents the transformation matrices are for the independent X, Y, and Z axes [2].

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Combined by multiplication, they form a general 3-axis transformation matrix[2]:

$$R_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}$$

$$R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix}$$

$$R = R_z(\alpha)R_y(\beta)R_x(\gamma)$$

$$= \begin{bmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{bmatrix}$$

The transformation matrix R however, transforms with the axis of reference to the world space origin. Therefore, a transformation is necessary to bring the vertex vertices $\vec{V}$ to the world origin, rotate them using R, and then send back to the location by adding $\vec{P}$.

$$\vec{V_R} = (R * (\vec{V} - \vec{P})) + \vec{P}$$

In implementation for the cube and tetrahedron generation:

Listing 4: Derived Objects generation with rotation matrices

```
    Matrix3D rotationMatrix = getRotationMatrix(
        angleInput*-1, angleInput*-2, angleInput * -1.5)
        ;

    int specular = 25;
    double reflective = 0;

    // Top Plane Coordinates
    Vector3D topCornerLR = (rotationMatrix * (Vector3D(
        center.getX()-width, center.getY()+width, center
        .getZ()-width) - center)) + center; // Top Left
        Rear Corner

    Repeat for each corner.


    // Top Triangles
    Primitive t1 = Primitive(color, topCornerLR,
        topCornerRR, topCornerLF, specular, reflective);
    Primitive t2 = Primitive(color, topCornerRR,
        topCornerLF, topCornerRF, specular, reflective);

    Repeat for each triangler

    array<Primitive, 12> triangles = {t1, t2, t3, t4, t5,
        t6, t7, t8, t9, t10, t11, t12};
    return triangles;
}

array<Primitive, 4> buildTetrahedron2(Vector3D color,
    Vector3D center, double width, int specular, double
    reflective, double angleInput){
    Matrix3D rotationMatrix = getRotationMatrix(
        angleInput*1, angleInput*2, angleInput*3);
```

```
    Vector3D pointA = (rotationMatrix * (Vector3D(center.
        getX(), center.getY(), center.getZ()-width) -
        center)) + center; // Tip

    Repeat for each point.

    Primitive t1 = Primitive(Vector3D(255, 0, 0), pointA,
         pointB, pointC, specular, reflective);

    Repeat for each triangle.

    array<Primitive, 4> triangles = {t1, t2, t3, t4};
    return triangles;
}
```

2.3 Positioning Primitives in Ray Tracer

When it comes to positioning the primitives in the ray tracer, this can be done simply by applying them a position vector $\vec{P}$, and the angle of rotation. But since the primitives must spin in a circle, a loop that pre-defines the position vectors is ran at the beginning of the program. Then, when it generates images, it will iterate through this new list of position vectors $\vec{P}_f$, and apply them to the scene. Since the meshes are equidistant, the total degree angle is divided by 3 and using modulo to prevent out of index errors. The index is selected by the current frame the program is running.

Listing 5: Code to use coordinates for the scene.

```
Scene scene = makeScene(circle[frame % 360], circle[((int
    )frame + 120) % 360], circle[((int)frame + 240) %
    360], frame);
```

The code below is for generating the coordinates.

Listing 6: Code for circle coordinate generation.

```
    array<Vector3D, 360> list;
    double x;
    double y;
    int angle;
    double rads;
    for (angle = 0; angle <= 360; angle++){
        rads = angle*(3.14/180);
        x = std::cos(rads);
        y = std::sin(rads);
        list[angle] = Vector3D(x*2, y*2, 8);
    }
    return list;

}
```

3 Parallelizing Ray Tracer

Rendering just a single frame can take a lot of time depending on how large the dimensions of the render is. It takes long to render just one frame, let alone over 360 frames. Whilst this could be left alone to render overnight, parallelizing the system would result in drastic time performance improvement which makes it a worthwhile implementation.

3.1 Theory and Implementation of Parallelization

Processors have the ability to run multiple instructions at the same time through the process of multi-threading. This segments the CPU into "partitions" to handle different independent processes[4].

Whilst a GPU would have been ideal, the time and knowledge constraints led to seeking a simple-to-implement system to parallelize the ray tracer which resulted in the use of OpenMP. OpenMP is simple to install and run in C++, especially when it comes to parallelizing for loops. Simply call the pragma function to set it up.

Listing 7: Getting coordinates.

```
#pragma omp parallel for
    for (int x = -CanvasWidth/2; x < CanvasWidth/2; x++){
        for (int y = -CanvasHeight/2; y < CanvasHeight/2;
            y++){...
```

A ray tracer is simple to parallelize given that each pixel for the scene "shoots" an independent ray to all the others. Each thread can handle all the calculations of a single pixel, and since the processes run at the same time, it results in faster rendering times for each frame.

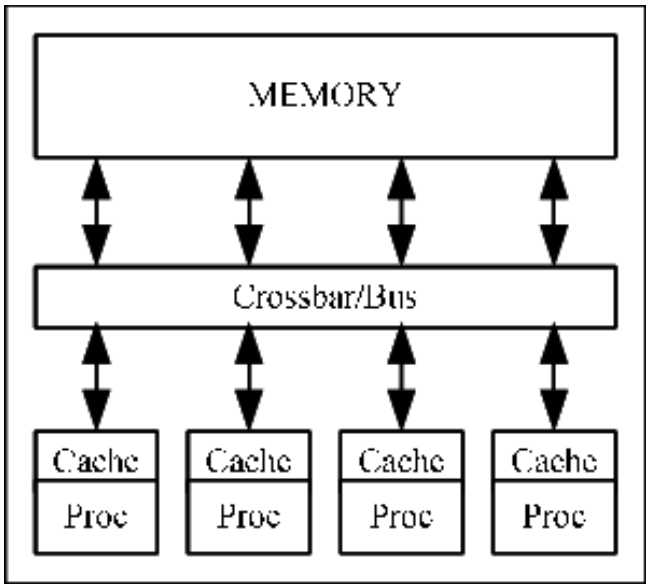

Figure 2: OpenMP parallel distribution [4].

OpenMP was not correctly implemented in the code resulting in "rogue pixels" throughout images. However, they're so small it's barely noticeable on high resolution images.

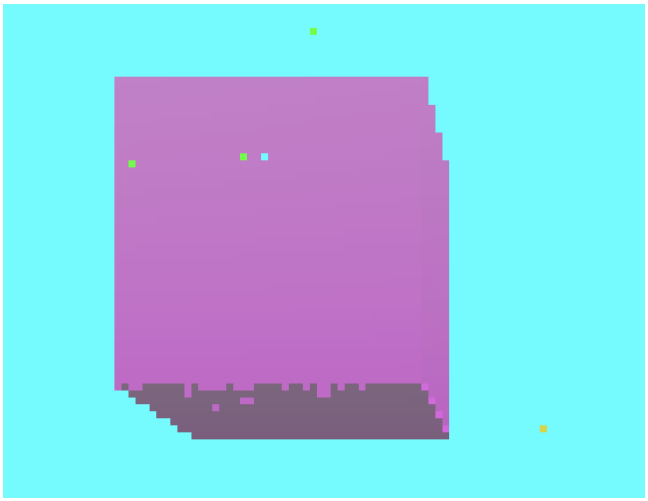

Figure 3: Rogue Pixels on low-resolution render of cube.

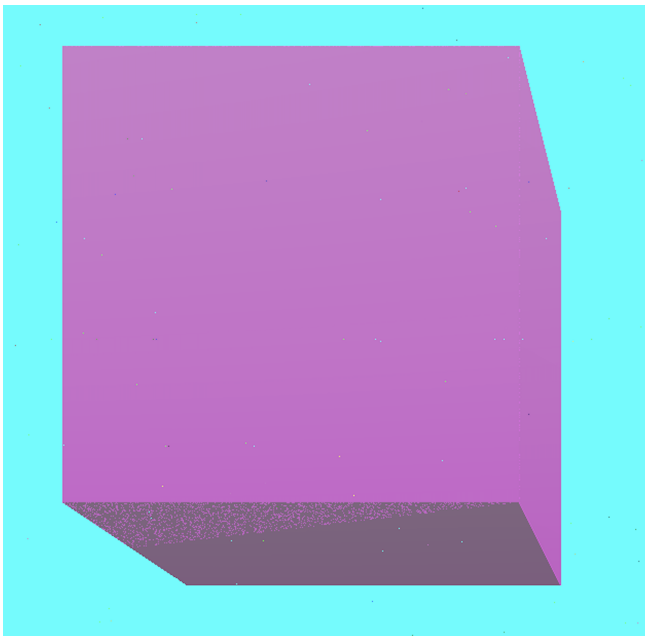

Figure 4: Rogue Pixels on high-resolution render of cube.

The performance benefits on the parallelization far outweight the problems of the rogue pixels. **Table 1** demonstrates the render time differences for different resolution renders of the same image using parallelization and not using parallelization.

Table 1: Render Times

Resolution (Pixels)	Non-Parallel Time (s)	Parallel Time (s)
500x500	45.10s	15.04s
800x800	117.47s	34.70s
1000x1000	181.19s	55.44s
4000x4000	3,000s	867.94s

There's **~70% time improvement**. When parallelized, the system takes 70% less time to render than in a non-parallelized system. This is useful for when it comes to rendering long sequences of images.

4 Applying Motion to the Ray Tracer

As explained in the introduction, animations consist of running frames one after the other in rapid succession resulting in "tricking" the brain to believe there is motion, despite what is being shown is a sequence of photos one after the other. Since the ray tracer already renders images, this makes this first step easy. The current Ray Tracer just needs a loop and some variables to change the positions of items in the scene.

4.1 Rendering Multiple Images

A simple loop in theory would result in rendering multiple images. However, within each frame, the 3D scene must be updated. This resulted in a major refactor of all the functions, and a new function *makeScene()* which will load all the primitives and lights with their locations into the 3D space before rendering.

4.2 Adding Motion

Within the loop of rendering frames, location and rotation variables are updated using the coordinates generated on the code **Listing 5**, then a new scene is made with *makeScene()* before calling *render()* to render the next frame. The transformations for the location and spin are applied to each primitive in the scene, the code of the primitives does the transformations to locate the primitives in the 3D space, and a new render is made. Each .ppm frame is named according to the frame number.

Listing 8: File naming.

```
fileName = "Output/animation/frame" + to_string(frame) +
    ".ppm";
```

4.3 Camera Movement

For camera movement, a dolly shot was considered for the beginning. However, the dolly needs to "slow down by the end to prevent a "jittered" movement. Maths proved to be useful here as an equation was developed to be fast at the beginning and gently slow down.

$$z_{camera} = 0.25 \cdot \left(\frac{360.0}{0.5 \cdot (\text{frame} + 1)} \right)$$

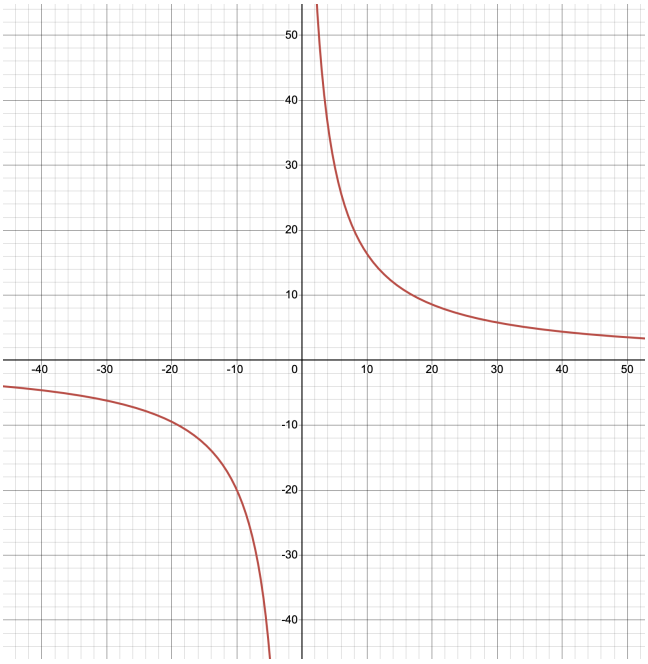

Figure 5: Function graph. Horizontal Axis = Frames, Vertical Axis = Z position.

The graph shows the position of the Z axis of the camera with respect to the frame number. It goes diminishing slowly the more frames are rendered, resulting in a smooth "slowdown" at the stop of the camera. Since it goes to infinity, a manual bound had to be added into code, this resulted in the camera stopping completely after a slowdown.

Listing 9: File naming.

```
result = 0.25*(360.0/(0.5*(frame + 1)));

        if (frame < 360 || result > 0.7){
            z_real = result;
        }
```

With this, the code generates a sequence of numbered frames which can then be sent to a video editor to compile into a video.

5 Conclusion

Although the principles explored in this project already exist in the real world, they are innovative features added to the simple ray tracer implemented in *Computer Graphics from Scratch* and are foundational to anything regarding Computer Graphics as translation and rotation are key elements in making 3-D models. And the animation principles are also invaluable when it comes to film-making and animation. These relatively simple algorithms and principles are what makes CG movies and models what they are today. Future additions could include new lighting and materials, better parallelization techniques and dynamic programming, and more advanced ray tracers like path tracers.

Acknowledgments

Special thanks to Dr. Edwin Flórez Gómez Professor of Mathematics and Computing at UPRM for the invaluable help to get me into the Computer Graphics class, explaining many complex topics from Calculus II to Computer Graphics, and mentoring my journey through Computer Graphics.

Thanks to Dr. Wilson Rivera Gallego Professor of Computer Science at UPRM and Dr. Omar Colón for approving the Computer Graphics course for my degree.

To my brother, Adhel-José for the invaluable help in consulting, the ideas, and support in making this system.

And thanks to God for giving me strength, endurance, and wisdom to accomplish this course and project.

References

[1] Gambetta, Gabriel. *Computer Graphics from Scratch*. No Starch Press, 2021.
[2] Wikipedia Contributors. *Rotation Matrix*. Wikimedia Foundation, 2019. https://en.wikipedia.org/wiki/Rotation_matrix.
[3] Science Behind Pixar. *Animation*. https://sciencebehindpixar.org/pipeline/animation.
[4] Kumar, Vipin. *Introduction to Parallel Computing*. Addison Wesley Longman, 1994.
[5] Owen, Scott. *Ray-Plane Intersection*. Siggraph.org, 1999. https://education.siggraph.org/static/HyperGraph/raytrace/rayplane_intersection.htm.

Received 13 December 2024; revised 12 December 2024; accepted TBD